BLM3011 - Operating Systems
Lecture Notes

Şafak Bilici

2020-2021

Introduction

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user.

- Hardware – provides basic computing resources
  - CPU, memory, I/O Devices
- Operating System
  - Controls and coordinates use of hardware among various applications and users.
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  - Text editors, compilers, web browsers, database systems, video games
- Users
  - People, machines, other computers

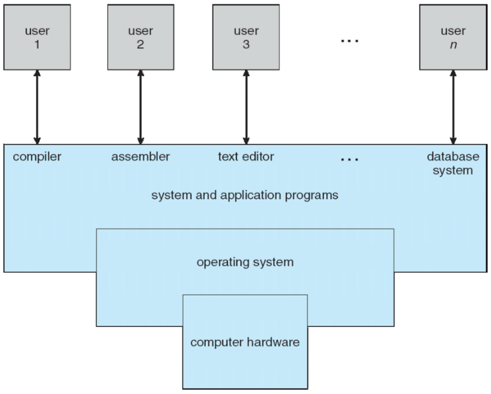


Figure 1: Computer System

A program that acts as an intermediary between a user of a computer and the computer hardware. Major functions of operating systems **may** include:

- Managing memory and other system resources.
- Imposing security and access policies.
- Scheduling and multiplexing processes and threads.
- Launching and closing user programs dynamically.
- Providing a basic user interface and application programmer interface.

Not all operating systems provide all of these functions. Single-tasking systems like MS-DOS would not schedule processes, while embedded systems like eCOS may not have a user interface, or may work with a static set of user programs.

An operating system is **not**

- The computer hardware.
- A specific application such as a word processor, web browser or game.
- A suite of utilities (like the GNU tools, which are used in many Unix-derived systems).
- A development environment (though some OSes, such as UCSD Pascal or Smalltalk-80, incorporate an interpreter and IDE).
- A Graphical User interface (though many modern operating systems incorporate a GUI as part of the OS).

While most operating systems are distributed with such tools, they are not themselves a necessary part of the OS. Some operating systems, such as Linux, may come in several different packaged forms, called distributions, which may have different suites of applications and utilities, and may organize some aspects of the system differently. Nonetheless, they are all versions of the same basic OS, and should not be considered to be separate types of operating systems.

What Operating System Do?

It depends on the point of view.

- Users want convenience, **ease of use** and **good performance**
    - Don't care about **resource utilization**
- But shared computer such as mainframe or minicomputer must keep all users happy
- Users of dedicate systems such as workstations have dedicated resources but frequently use shared resources from servers
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

OS is a **resource allocator**. It manages all resources and decides between conflicting requests for efficient and fair resource use. Also OS is an **control program**. It controls execution of programs to prevent errors and improrer use of the computer.

The "one program running at all times on the computer" is the **kernel**. The kernel of an operating system is something you will never see. It basically enables any other programs to execute. It handles events generated by hardware (called interrupts) and software (called system calls), and manages access to resources. The kernel usually defines a few abstractions like files, processes, sockets, directories, etc. which correspond to an internal state it remembers about last operations, so that a program may issue a session of operation more efficiently.

Computer Startup

A **bootstrap program** is loaded at power-up or reboot. Typically stored in ROM or EPROM, generally known as **firmware**. It initializes all the aspects of the system, loads operating system kernel and starts execution.

Computer-System Organization

A modern general-purpose computer system consists of one or more CPU s and a number of device controllers connected through a common **bus** that provides access between components and shared memory.

Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached.

A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. CPU moves data from/to main memory to/from local buffers.

Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

Device controller informs CPU that it has finished its operation by causing an **interrupt**.

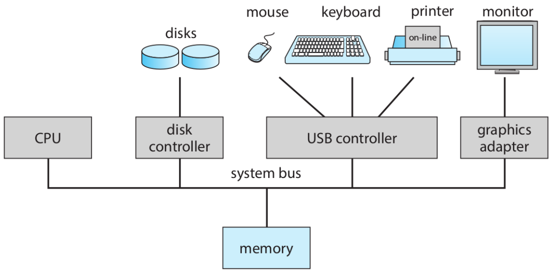


Figure 2: A typical PC computer system.

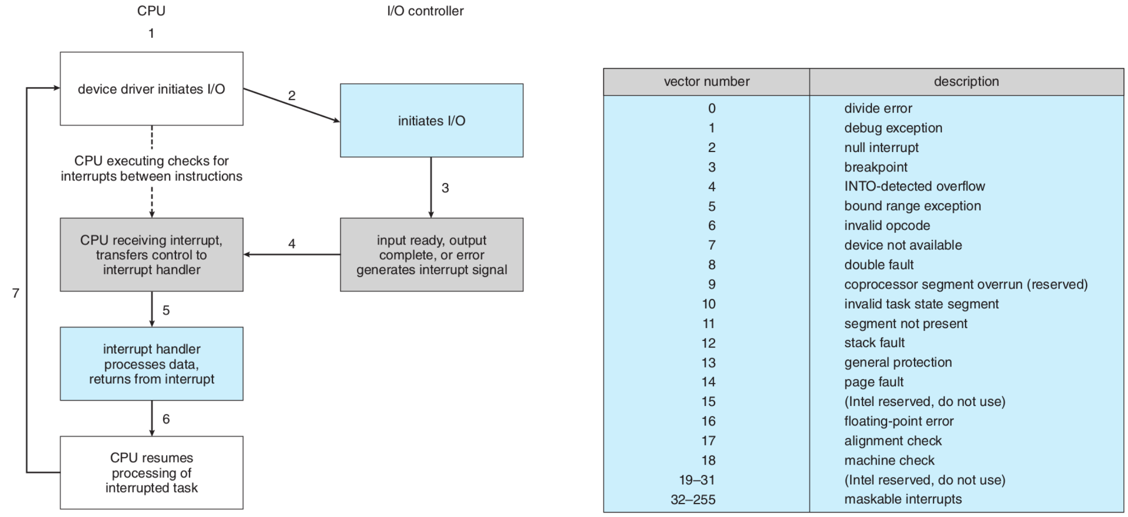
Interrupts

Consider a typical computer operation: a program performing I/O . To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

Hardware may trigger an interrupt at any time by sending a signal to the CPU , usually by way of the system bus. Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines. Interrupt architecture must save the address of the interrupted instruction. A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request. An operating system is

interrupt driven.


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

(a) Interrupt-driven I/O cycle.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

(b) Intel processor event-vector table.

Figure 3

The CPU hardware has a wire called the **interrupt-request** line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index.

We say that the device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- A trap or exception is a software-generated interrupt caused either by an error or a user request.
- An operating system is interrupt driven.

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- Determines which type of interrupt has occurred:
  - **polling**
  - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt.

Storage Structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called random-access memory, or RAM). Mainmemory commonly is implemented in a semiconductor technology called dynamic random-access memory (DRAM).

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a **bootstrap** program ,which then loads the operating system. Since **RAM is volatile** — loses its content when power is turned off or otherwise lost — we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically Erasable Programmable Read-Only Memory (EEPROM) and other forms of **firmware** —storage that is infrequently written to and is nonvolatile. EEPROM can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU,whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

Most computer systems provide secondary storage as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage devices are **hard-disk drives (HDDs)** and **nonvolatile memory (NVM)** devices, which provide storage for both programs and data.

In a larger sense, however, the storage structure that we have described is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes —to store backup copies of material stored on other devices, for example— are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

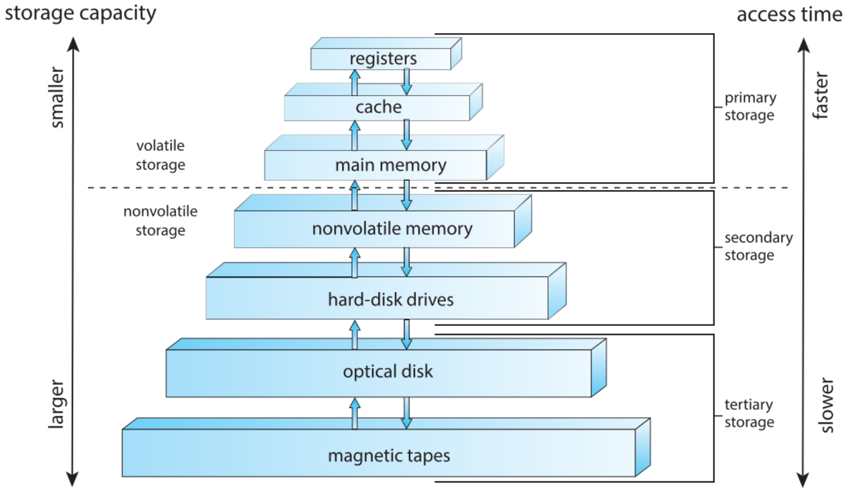


Figure 4: Storage-device hierarchy.

The top four levels of memory in the figure are constructed using semi-conductor memory, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. Themost common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software).
- Information in use copied from slower to faster storage temporarily.
- Faster storage (cache) checked first to determine if information is there.
  - If it is, information used directly from the cache (fast).
  - If not, data copied to cache and used there.
- Cache smaller than storage being cached.
  - Cache management important design problem.
  - Cache size and replacement policy.

Direct Memory Access Structure (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds.
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.
- Only one interrupt is generated per block, rather than the one interrupt per byte.

I/O Structure

A large portion of operating system code is dedicated to managing I/O,both because of its importance to the reliability and performance of a system and because of the varying nature of the devices. The form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

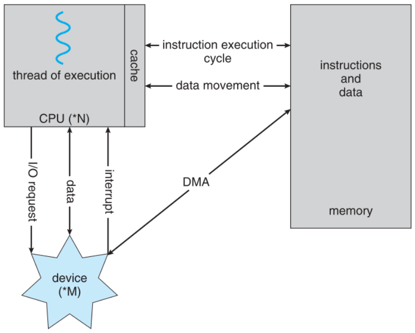


Figure 5: How a modern computer system works.

- After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt.
  - Wait loop (contention for memory access).
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
  - **System call** – request to the OS to allow user to wait for I/O completion.
  - **Device-status table** - contains entry for each I/O device indicating its type, address, and state.
  - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

Computer-System Architecture

Most systems use a single general-purpose processor, most systems have special-purpose processors as well. On modern computers, from mobile devices to servers, **multiprocessor** (also known as **parallel systems, tightly-coupled systems**) systems now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; it is less than N.

The most common multiprocessor systems use **symmetric multiprocessing (SMP)**, in which each peer CPU processor performs all tasks, including operating-system functions and user processes.

Asymmetric Multiprocessing is each processor is assigned a specie task.

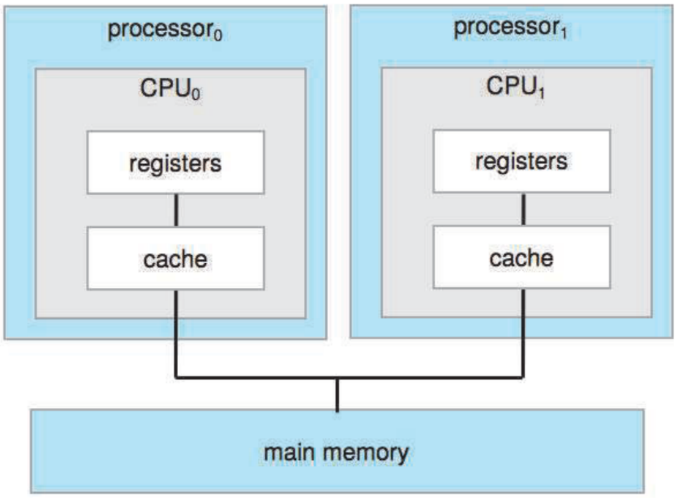


Figure 6: Symmetric multiprocessing architecture.

The definition of multiprocessor has evolved over time and now includes multicore systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

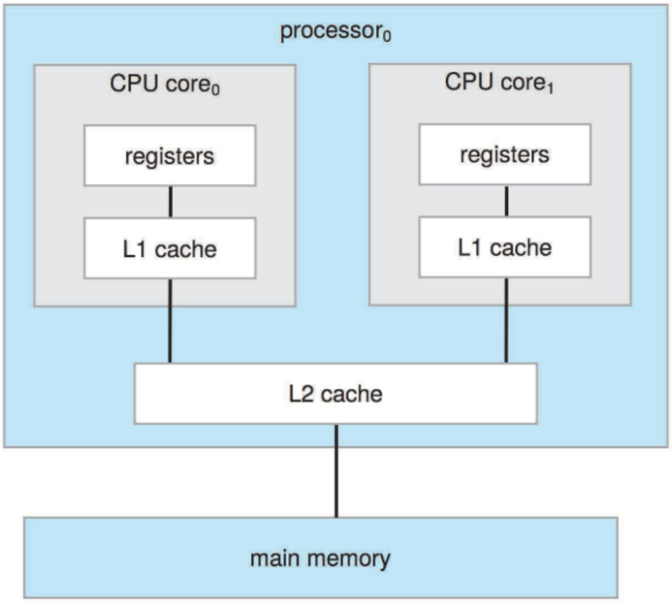


Figure 7: A dual-core design with two cores on the same chip.

Clustered Systems

- Like multiprocessor systems, but multiple systems working together.
  - Usually sharing storage via a **storage-area network (SAN)**.
  - Provides a **high-availability** service which survives failures
    - **Asymmetric clustering** has one machine in hot-standby mode.
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other.
  - Some clusters are for **high-performance computing (HPC)**.
    - Applications must be written to use **parallelization**.
  - Some have **distributed lock manager (DLM)** to avoid conflicting operations.

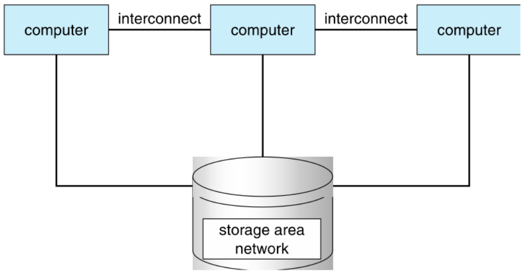


Figure 8: General structure of a clustered system.

Multiprogramming and Multitasking One

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically want to run more than one program at a time as well. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a **process**. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When that process needs to wait, the CPU switches to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**.

- **Multiprogramming (Batch system)** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times.
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute.
  - A subset of total jobs in system is kept in memory.
  - One job selected and run via **job scheduling**.
  - When it has to wait (for I/O for example), OS switches to another job.
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing.
  - **Response time** should be $<$ 1 second
  - Each user has at least one program executing in memory → **process**.
  - If several jobs ready to run at the same time → **CPU scheduling**.
  - If processes don't fit in memory, **swapping** moves them in and out to run.
  - **Virtual memory** allows execution of processes not completely in memory.

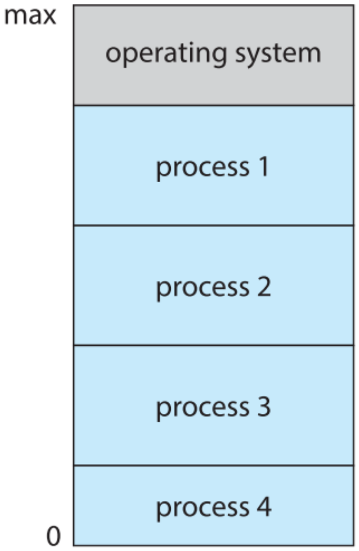


Figure 9: Memory layout for a multiprogramming system.

Transition from User to Kernel Mode

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs —or the operating system itself—to execute incorrectly. At the very least, we need two separate modes of operation: **user mode** and **kernel mode** (also called supervisor mode, system mode,or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).

- Timer to prevent infinite loop / process hogging resources.
  - Timer is set to interrupt the computer after some time period.
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction).
  - When counter zero generate an interrupt.
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time.

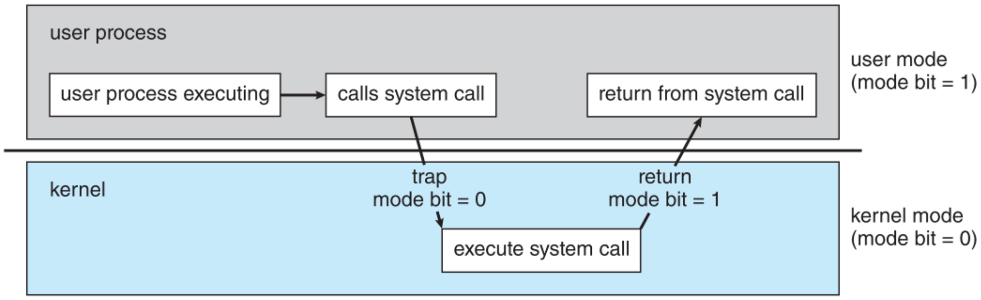


Figure 10: Transition from user to kernel mode.

Resource Management

Process Management

A program can do nothing unless its instructions are executed by a CPU.A program in execution, as mentioned, is a process.Aprogram such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process.

Aprocess needs certain resources—including CPU time,memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along.

- A process is a program in execution. It is a unit of work within the system. Program is a **passive entity** process is an **active entity**.
- Process needs resources to accomplish its task.
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources.
- Single-threaded process has one **program counter** specifying location of next instruction to execute

- Multi-threaded process has one program counter per thread.
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs.
    - Concurrency by multiplexing the CPUs among the processes / threads.

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication.
- Providing mechanisms for deadlock handling

Memory Management

- To execute a program all (or part) of the instructions must be in memory.
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when.
    - Optimizing CPU utilization and computer response to users.
- Memory management activities
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes (or parts thereof) and data to move into and out of memory.
    - Allocating and deallocating memory space as needed.

Storage Management

- OS provides uniform, logical view of information storage
    - Abstracts physical properties to logical storage unit: **file**
    - Each medium is controlled by device (i.e., disk drive, tape drive)
        - Varying properties include access speed, capacity, data- transfer rate, access method (sequential or random)
- File-System management
    - Files usually organized into directories
    - Access control on most systems to determine who can access what
    - OS activities include
        - Creating and deleting files and directories
        - Primitives to manipulate files and directories
        - Mapping files onto secondary storage
        - Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance.
- Entire speed of computer operation hinges on disk subsystem and its algorithms.
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications
  - Varies between WORM (write-once, read-many-times) and RW (read-write).

WEEK 4

Operating-System Structures

Operating System Services

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs.

- **User Interface**:Almost all operating systems have a **user interface (UI)**. This interface can take several forms. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.
- **Program Execution**: The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error).
- **I/O Operations**: Arunning program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system Manipulation**: The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications**: There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications

may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

- **Error Detection**: The operating system needs to be detecting and correcting errors constantly. Errorsmay occur in theCPUandmemoryhardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.
- **Resource Allocation**: When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may havemuchmore general request and release code.
- **Accounting**: To keep track of which users use how much and what kinds of computer resources (different from log files -utmp-).
- **Protection and Security**: The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.

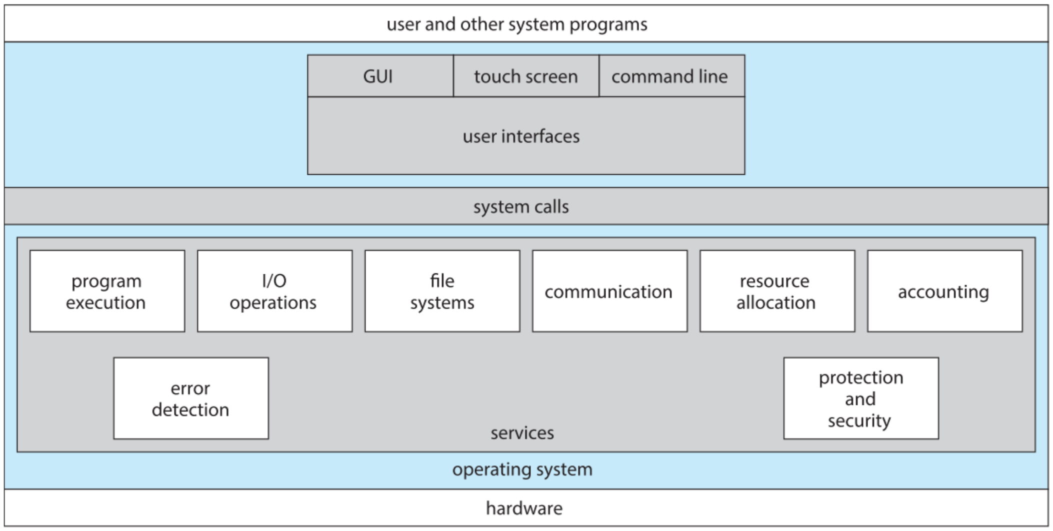


Figure 11: A view of operating system services.

User and Operating-System Interface

Command Interpreters (CLI)

Most operating systems, including Linux, UNIX, and Windows, treat the com- mand interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX andLinux systems, a user maychoose among sev- eral different shells, including the Cshell, Bourne-Again shell (bash), Korn shell, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference.

- Sometimes implemented in kernel, sometimes by systems program.
- Sometimes multiple flavors implemented: **shells**.
- Primarily fetches a command from user and executes it.
- Sometimes commands built-in, sometimes just names of programs.

Graphical User Interface (GUI)

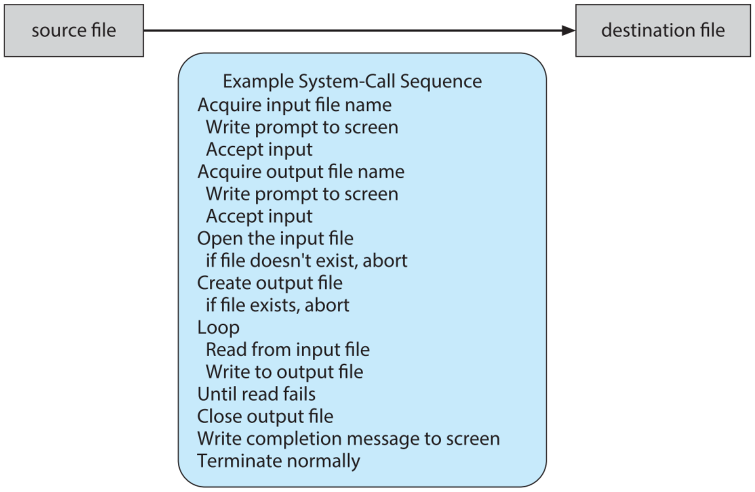
Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and-menu system characterized by a desktop metaphor. The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

- Invented at Xerox PARC.
- Microsoft Windows is GUI with CLI "command" shell.
- Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available.
- Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME).

System Calls

System calls provide an interface to the services made available by an operat- ing system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM).



```
Example System-Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Figure 12: Example of how system calls are used: `cp in.txt out.txt`

System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers.
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values.
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call.
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler).

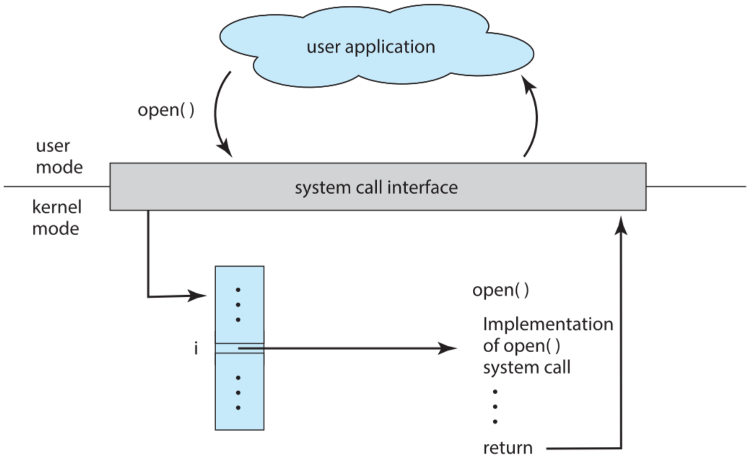


Figure 13: Example of how system calls are used: `cp in.txt out.txt`

Whywould an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application program- mer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register. Linux uses a combination of these approaches. If there are five or fewer parameters, registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or **pushed**, onto a **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

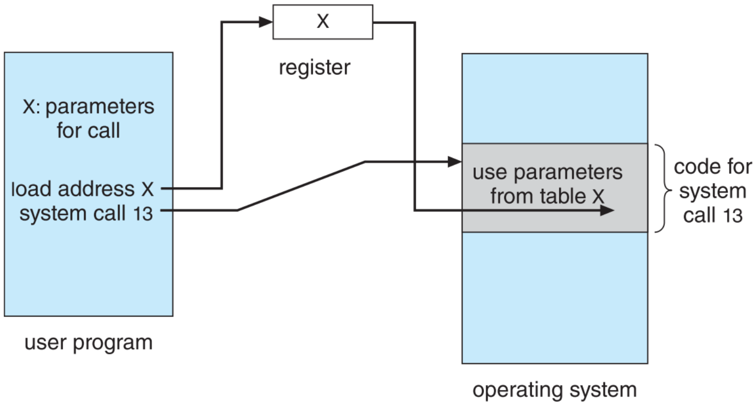


Figure 14: Example of how system calls are used: `cp in.txt out.txt`

Types Of System Calls

System calls can be grouped roughly into six major categories: **process control**, **fil management**, **device management**, **information maintenance**, **communi- cations**, and **protection**.

- **Process Control**
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error

  - **Debugger** for determining **bugs**, single step execution
  - **Locks** for managing access to shared data between processes

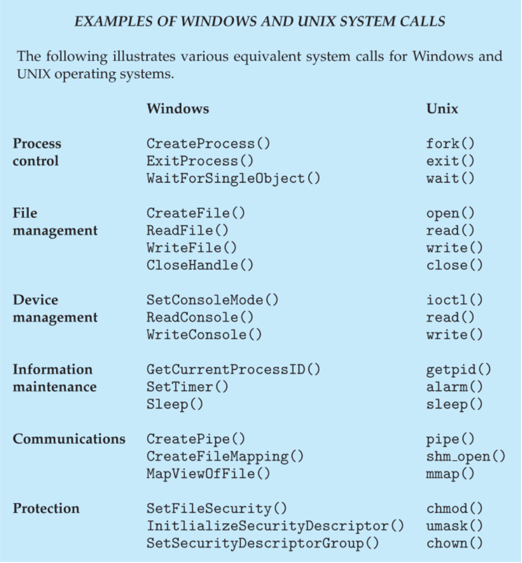
- **File Management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

- **Device Management**
  - request device, reselase device
  - read, write, reposition
  - get device attributes, set device attribures
  - logically attach or detach devices

- **Information Maintenance**
  - get time or date, set time or date
  - get system data, set systemd ata
  - get and set process, file or device attributes

- **Communications**
  - create, delete commumication connection
  - send, receive message if message passing model to host name or process name
    - from client to server
  - transfer status informatiom
  - attach and detach remote devices

- **Protection**
  - control access to resources
  - get and set permissions
  - allow and deny user access



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	**Unix**

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

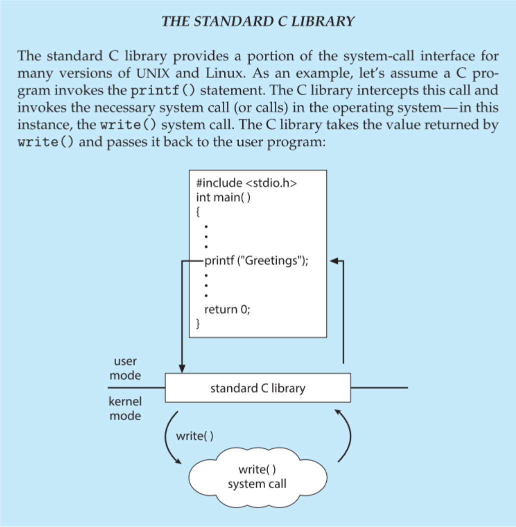
Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

(a) Examples of Windows and Unix System Calls.



THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

```
#include <stdio.h>
int main( )
{
  .
  .
  .
  printf ("Greetings");
  .
  .
  .
  return 0;
}
```

(b) printf().

Figure 15

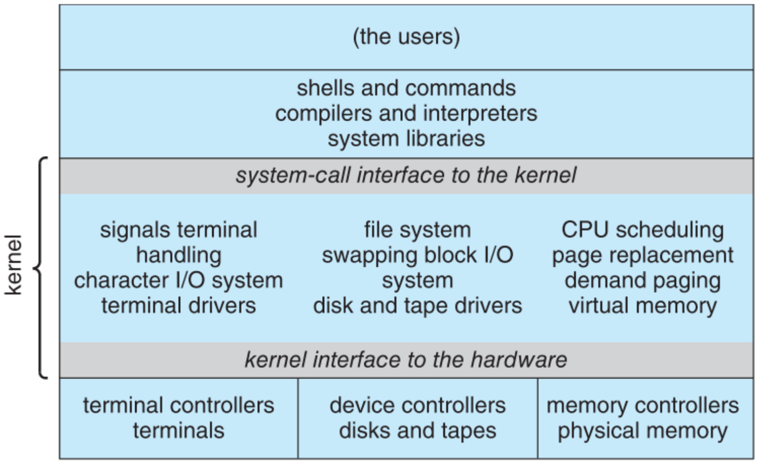


Figure 16: Traditional UNIX system structure.

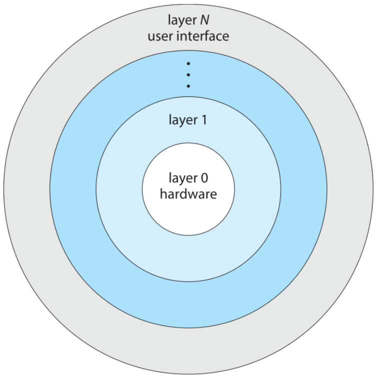


Figure 17: Layered Approach.

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

Microkernel System Structure

We have already seen that the original UNIX system had a monolithic struc- ture. As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the micro- kernel approach. This method structures the operating system by removing all non-essential components from the kernel and implementing them as user- level programs that reside in separate address spaces. The result is a smaller kernel. The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.

- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)

    - More secure

- Detriments:
    - Performance overhead of user space to kernel space communication.

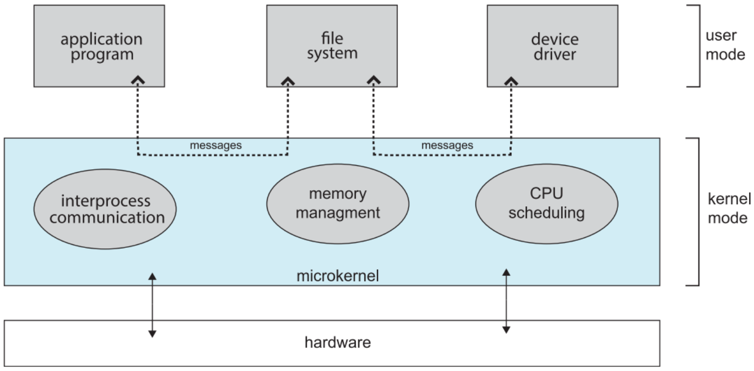


Figure 18: Architecture of a typical microkernel.

Modules

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules (LKMs)**. Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows. The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

- Uses object-oriented approach.
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

Hybrid Systems

In practice, very few operating systems adopt a single, strictly defined struc- ture. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.

- Hybrid combines multiple approaches to address performance, security, usability needs
- Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality.
- Windows mostly monolithic, plus microkernel for different subsystem personalities

Operating System Generation

Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.

SYSGEN program obtains information concerning the specific configuration of the hardware system. It is ussed to build system-specific compiled kernel or system-tuned. Can general more efficient code than one general kernel.

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot** block at fixed location loaded by ROM code, which loads bootstrap loader from disk.
  - Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options

WEEK 5 & WEEK 6

Process Concept

- An operating system executes a variety of programs:
  - Batch system: **jobs**
  - Tİme-shared systems - **user programs** or **tasks**
- **Process**: a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during runtime.
- Program is **active** entitiy stored on disk (**executable file**), process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI Mouse click, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

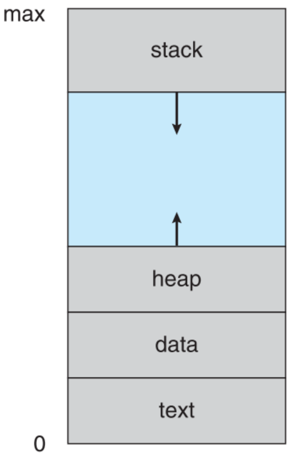
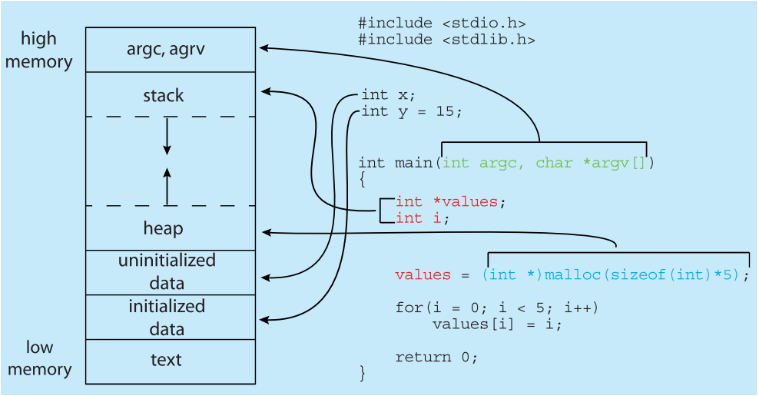


Figure 19: Layout of a process in memory.



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

   values = (int *)malloc(sizeof(int)*5);

   for(i = 0; i < 5; i++)
       values[i] = i;

   return 0;
}
```

Figure 20: Layout of a process in memory.

Process States

- As a process executes, it changes state
  - **New**: The process is beaing created.
  - **Running**: Instructions are being executed.
  - **Waiting**: The process is waiting for some event to occur.
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Terminated**: The process has finished execution.

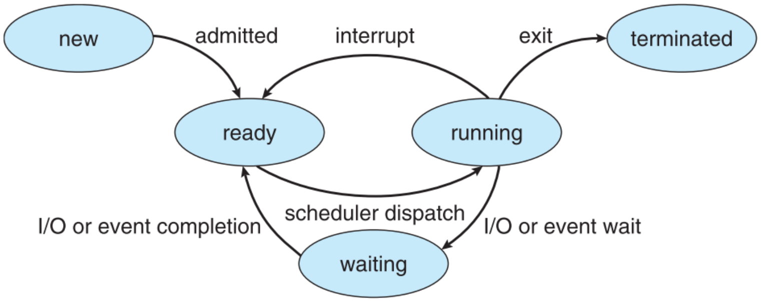


Figure 21: Diagram of process state.

Process Control Block (PCB)

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. It contains many pieces of information associated with a specific process, including these:

- **Process State**: The state may be new, ready, running, waiting, halted, and so on.
- **Program Counter**: The counter indicates the address of the next instruction to be executed for this process.
- **CPU Registers**: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code informa- tion. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU-Scheduling Information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-Management Information**: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting Information**: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O Status Information**: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

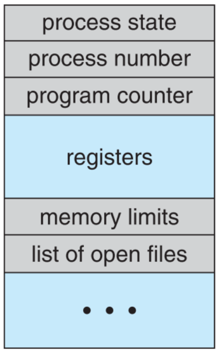


Figure 22: Process control block (PCB)

CPU Switch From Process to Process

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching.

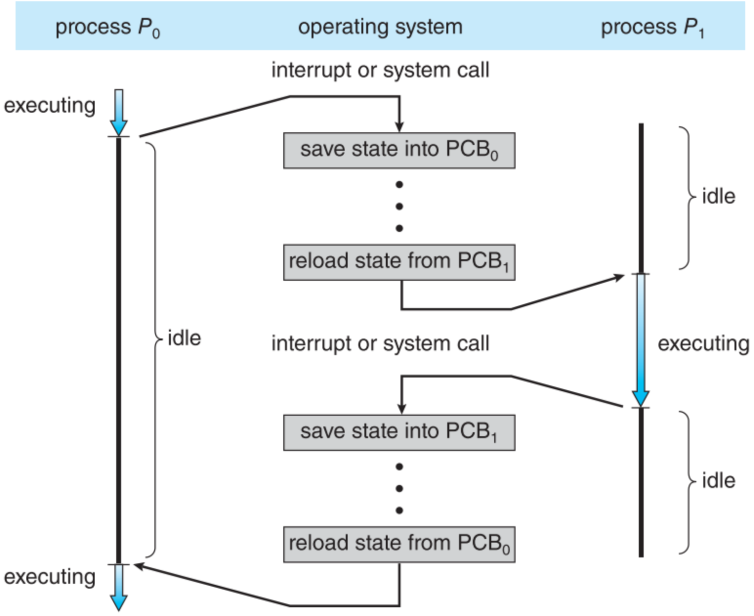


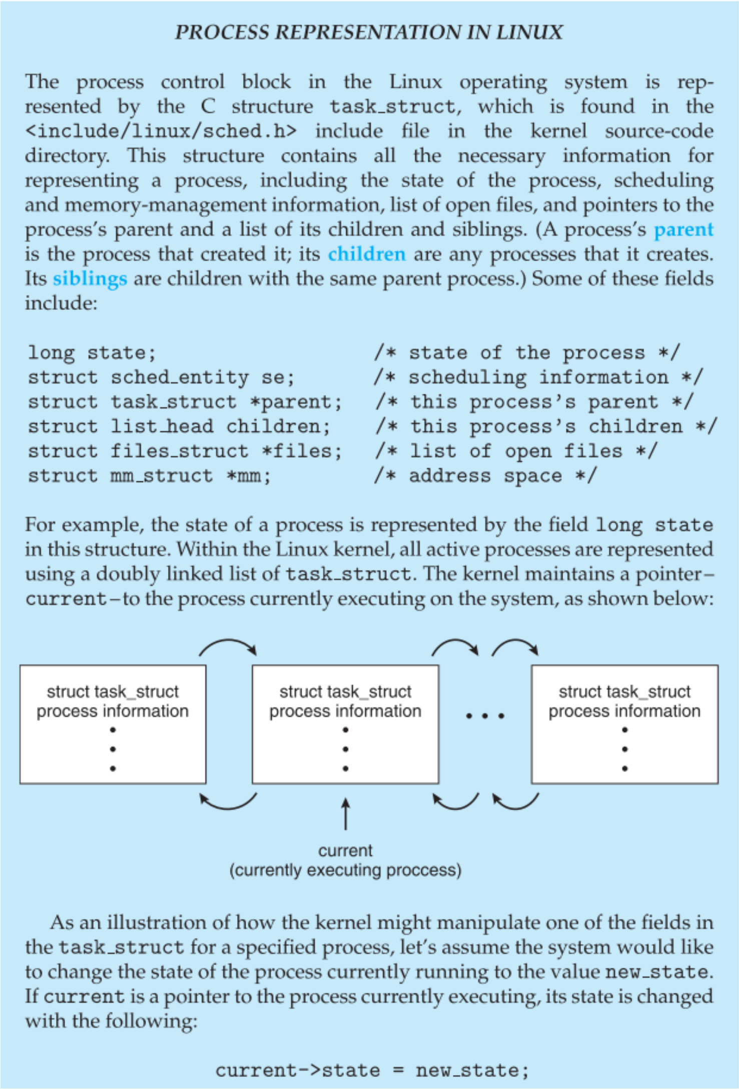
Figure 3.6 Diagram showing context switch from process to process.

Figure 23: Diagram showing context switch from process to process.

Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control: **threads**
- Must then have storage for thread details, multiple program counters in PCB.



PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<include/linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's **parent** is the process that created it; its **children** are any processes that it creates. Its **siblings** are children with the same parent process.) Some of these fields include:

```
long state;                    /* state of the process */
struct sched_entity se;        /* scheduling information */
struct task_struct *parent;    /* this process's parent */
struct list_head children;     /* this process's children */
struct files_struct *files;    /* list of open files */
struct mm_struct *mm;          /* address space */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer– `current`–to the process currently executing on the system, as shown below:

As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

Figure 24: Process representation in linux.

Process Scheduling

As processes enter the system, they are put into a ready queue,wherethey are ready and waiting to execute on a CPU's core This queue is generally stored as a linked list; a **ready-queue** header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

- Maximize CPU use, quickly switch processes onto CPU for time sharing.
- **Process scheduler** selects among available processes for next execution on CPU next execution on CPU.
- Maintains **scheduling queues** of processes
  - ○ **Job Queue**: set of all processes n the system
  - ○ **Ready Queue**: set of all processes residing in main memory, ready and waiting to execute
  - ○ **Device Queues**: set of processes waiting for an I/O device

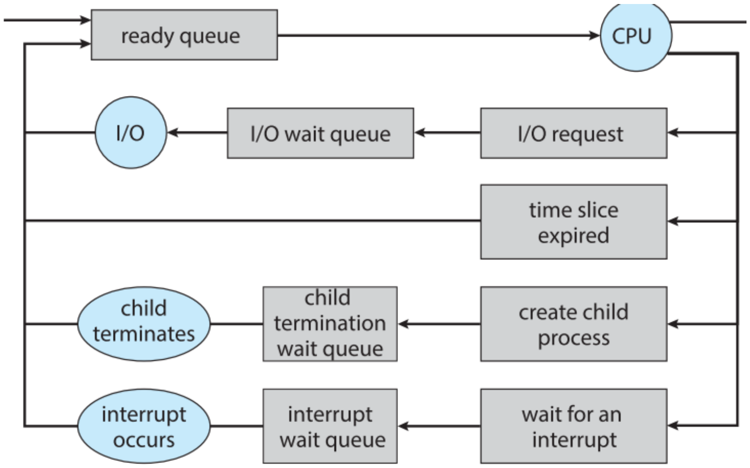


Figure 25: Queueing-diagram representation of process scheduling.

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU.
    - Sometimes the only scheduler in a system.
    - Short-term scheduler is invoked frequently (must be fast)
- **Long-term scheduler** (or **job scheduler**) - selects which process should be brought into the ready queue.
    - Long-term scheduler is invoked infrequently (may be slow)
    - long-term scheduler controls the **degree of multiprogramming**
- Processes can be desribed as either
    - **I/O bound process**: spends more time doing I/O than computations, many short CPU bursts.
    - **CPU-bound process**: spends more time doing computations; few very long CPU bursts

Medium-term scheduler can be added if degree of multiple programming needs to decrease. Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**.

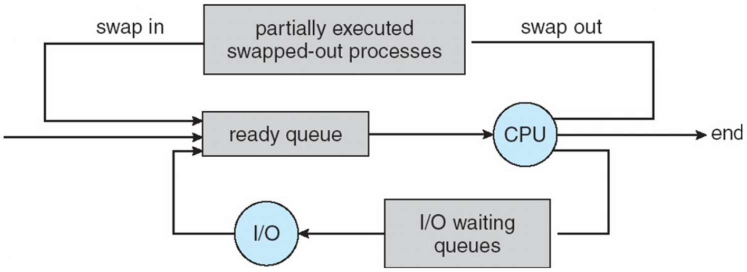


Figure 26: Swap in-out.

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**.
- **Context** of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes.
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

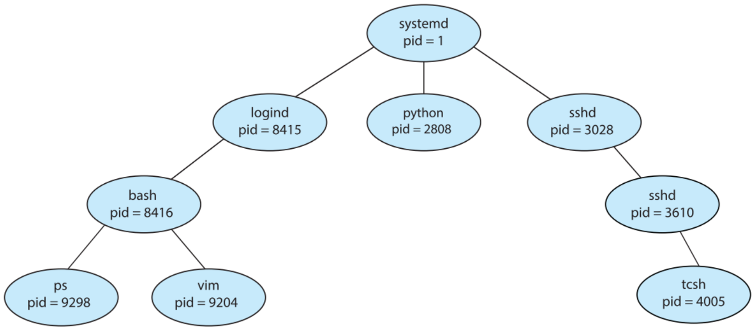


Figure 27: A Tree of Processes in Linux

- Address space
  - Child has a program loaded into it
  - Child duplicate of parent
- UNIX:
  - **fork()** system call creates new process.
  - **exec()** system call used after a fork() to replace the process' memory space with a new program

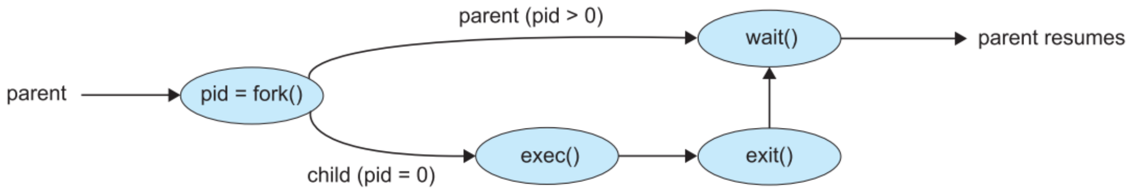


Figure 28: Process creation using the fork() system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){

  pid_t pid;
  pid = fork();

  if(pid<0){ // error occured
    fprintf(stderr,”Fork␣failed”);
    return −1;
  }
  else if(pid==0){ //child process
    execlp(”/bin/ls”,”ls”,NULL);
  }
  else{ // parent process
    wait(NULL); // parent will wait for the child to complete
    printf(”Child␣Complete”);
  }
  return 0;
}
```

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination**. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process.
  pid = wait(&status);
- If no parent waiting (did not invoke wait()) process is a **zombie**
- If parent terminated without invoking **wait** , process is an orphan.

Interprocess Communication (IPC)

- Processes within a system may be **independent** or **cooperating**.
- Cooperating process can affect or be affected by other processes, including data sharing
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**.
- Two models of IPC
  - **Shared Memory**
  - **Message Passing**

(a) Message passing. (b) shared memory.

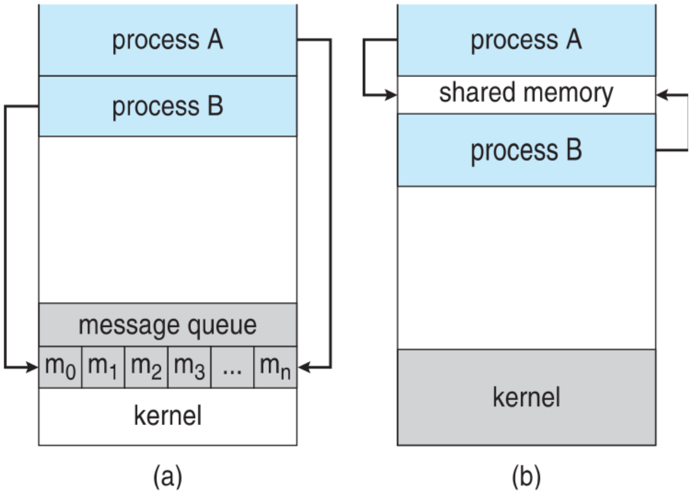


Figure 29: Process creation using the fork() system call.

Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Shared Memory

- An area of memory shared among the processes that wish to communicate.
- The communication is under the control of the users processes, not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations: **send**(message) and **receive**(message)
- The message size is either fixed or variable
- If processes P and Q wish to communicate, they need to:
  - Establish a **communication** link between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?
- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- Properties of communication link:
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A
- Mailbox sharing
  - P1 , P2 , and P3 share mailbox A
  - P1 , sends; P2 and P3 receive
  - Who gets the message?
- Solutions:
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** – the sender is blocked until the message is received
  - **Blocking receive** – he receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** – the sender sends the message and continue
  - **Non-blocking recieve** – the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/shm.h>
#include<sys/stat.h>
#include <sys/mman.h>

 int main(){

   const int SIZE = 4096; // the size (in bytes) of shared memory
   const char* name = ”OS”; // name of the shared memory object
   const char* message_0 = ”Hello”;
   const char* message_1 = ”␣World!”;

   int shm_fd; // shared memory file descriptor
   void *ptr; // pointer to shared memory object

   shm_fd = shm_open(name,O_CREAT|O_RDWR, 0666);
   //create the shared memory object
   ftruncate(shm_fd, SIZE);
   // configure the size of the shared memory object
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
   // memory map the shared memory object

   sprintf(ptr,”%s”,message_0);
   ptr += strlen(message_0);
   sprintf(ptr,”␣%s”,message_1);
   ptr += strlen(message_1);

   return 0;
 }
```

Consumer:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<sys/shm.h>
#include<sys/stat.h>
#include<fcntl.h>

 int main(){

   const int SIZE = 4096; // the size in bytes of shared memory object
   const char* name = ”OS”; // name of the shared memory object
   int shm_fd; // shared memory fd
   void *ptr; //pointer to shared memory object

   shm_fd = shm_open(name,O_RDONLY,0666); /
   / open the shared memory object
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd,0);
   // memory map the shared memory object
   printf(”%s\n”,(char*)ptr); // read from the shared memory object
   shm_unlink(name); //remove the shared memory object

   return 0;}
```

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are well known, used for standard services.
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running.

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipe

Producer writes to one end (the write-end of the pipe) and Consumer reads from the other end (the read-end of the pipe). Ordinary pipes are therefore unidirectional

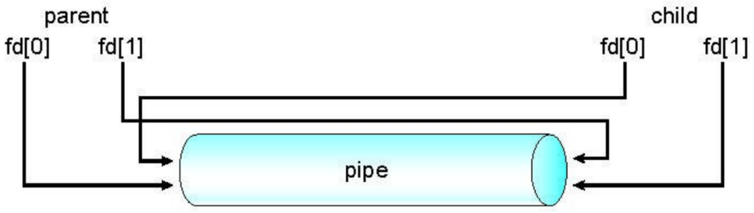


Figure 30: Ordinary pipe.